Android programming in Free Pascal: Networking, external code and threads

Michaël Van Canneyt

September 4, 2015

Abstract

This article will explain how to access the network (internet) from an Android application, to fetch some external data. Accessing the network in Android is best done in a separate thread, so the concept of async tasks is also explained. Lastly, the external data comes in JSON format.To parse this JSON, some native Java code will be incorporated in the application: the article will show how to do this as well.

1 Introduction

In previous articles, some techniques for Android programming were discussed. Database access, application preferences. Most, if not all, applications that run on an Android device will need access to the internet at some point. The Android SDK contains some classes that make this a fairly easy task - basically it contains the equivalent of the `TFPHttpClient` class from Free Pascal or the `THTTPSend` class present in Synapse.

Since accessing the network (or internet) and waiting for a response is a fairly time-consuming task, it is best performed in the background (actually, this is a practice recommended by Google). Again, Android offers a class to make this an easy exercise.

To demonstrate this, we will again expand the application presented in previous articles: the application to note absenteism is expanded to fetch the list of pupil groups and pupils from a REST service somewhere on Internet.

The REST service returns JSON. The JSON is parsed using a Java library: this java library will be accessed from Pascal code. The integration of this Java library in the application will also be discussed.

When executed, these various operations will take some time, So it is good to report back to the user to let him know the application is still working. To this end, the article will also discuss the use of various Android dialogs.

2 Accessing the network

Before the network can be accessed, it is useful to check whether the network can be accessed: the device may be without active network connection, in which case network access will result in an error. Fortunately, the Android API offers a simple function to determine whether there is access to a network.

The Android environment has several system services. One of these services - the connectivity service - controls access to internet. The Android API has a class that can be used to query this service: `android.net.ConnectivityManager`. An instance of this class can be obtained directly from the application's context. This allows us to write the following simple function:

```
Function HaveInternetConnection(AContext : ACContext): Boolean;

Var
  connMgr : ANConnectivityManager;
  Info : ANNetworkInfo;

begin
  connMgr:=ANConnectivityManager(
             AContext.getSystemService(ACContext.CONNECTIVITY_SERVICE)
           );
  info:=connMgr.getActiveNetworkInfo();
  Result:=Assigned(Info) and (Info.isConnected()=True);
end;
```

The `getSystemService` allows to retrieve instances of various system classes, they are identified by a name. The `ACContext` class contains some class constants with the names of system services: `CONNECTIVITY_SERVICE` is the service we need. The `ANConnectivityManager` class contains some methods to query connectivity: `getActiveNetworkInfo` returns information on the active network (if there is any). The last line of the function checks whether the network is actually connected.

If there is no connection, one of the ways to get a connection is to activate Wi-Fi. This can be done with another system service, namely the WIFI service, which works in much the same way as the network service:

```
Procedure TAbsenteeDBHelper.EnableWIFI(AContext : ACContext);

Var
  wifiMgr: ANWWifiManager;

begin
  wifiMgr:=ANWWifiManager(
            Acontext.getSystemService(ACContext.WIFI_SERVICE)
           );
  wifiMgr.setWifiEnabled(True);
end;
```

There are multiple ways to connect to the internet: Mobile networking, WIFI etc. In general, it is probably a better choice to instruct the user to enable some form of networking.

Once internet connectivity is established, the application can proceed to download some information from internet.

There are 3 classes involved in this process:

JNURL A class that represents an URL: any network location. It supports several common internet protocols (URL schemes), such as HTTP, HTTPS or FTP.

JNHttpURLConnection A class that represents an HTTP connection. A class of this type must be instantiated through a `JNURL` instance. To use this class, the application manifest needs to include the permission `android.permission.INTERNET`. When a user installs the application, the user will then be presented with a dialog that notifies him (or her) that the application needs internet access.

JIInputStream A Java IO class which is the equivalent of a `TStream` in Object Pascal. The output from the `JNHttpURLConnection` request is accessible using a stream of this type.

The method to download an URL and convert the response to a string can be written using these 3 classes, in a new auxilary class:

```
Function TDownloadHelperTask.downloadUrl(AURL: String): String;

Var
   url : JNURL;
   Conn : JNHttpURLConnection;
   Ins : JIInputStream;

begin
  ins:=Nil;
  URL:=Nil;
  Conn:=Nil;
  try
    // Create URL
    url:=JNURL.Create(JLString.Create(AURL));
    // Request connection
    conn:=url.openConnection() as JNHttpURLConnection;
    // Set some parameters.
    conn.setReadTimeout(10000);
    conn.setConnectTimeout(15000);
    // We want to read
    conn.setDoInput(true);
    conn.setRequestMethod('GET');
    // Go !
    conn.Connect();
    // Check response status.
    if (conn.getResponseCode()=200) then
      begin
      ins:=conn.getInputStream();
      Result:=ReadAsString(ins);
      end
    else
      Result:='';
  finally
    if (ins<>nil) then
      ins.close();
  end;
end;
```

There is nothing magical in this function. The `setReadTimeOut` and `setConnectTimeOut` are self-explaining: they set timeout parameters, these avoid that the class will wait forever for a response. The `setDoInput` tells the connection object that we want to read data from it, and the `SetRequestMethod('GET')` tells the object that the HTTP method GET must be executed. Finally, the `Connect` method actually executes the request.

The response is returned in the form of a `JIInputStream` class, which is a `TStream` equivalent in Java: a stream of bytes. This stream of bytes is converted to a string using the `ReadAsString` method (which we must create ourselves). It performs a conversion of the contents of the stream to a String. This method uses 2 auxiliary classes:

JIBufferedReader A buffered reader. This class allows to read text lines from a stream, using an intermediate buffer for speed improvement.

JLStringBuilder A Java equivalent of the `TStringBuilder` class in Delphi and .NET.

Using these classes, the following straightforward code will then read the content of a stream line by line using the buffered reader, and converts the lines to a single string with the string builder class;

```
function TDownloadHelperTask.ReadAsString(
                ins: JIInputStream): String;

Var
  Reader : JIBufferedReader;
  SB : JLStringBuilder;
  L : JLString;

begin
  Reader:=JIBufferedReader.Create(
           JIInputStreamReader.Create(ins,'UTF-8')
          );
  SB:=JLStringBuilder.Create;
  Result:='';
  L:=Reader.readLine;
  while L<>Nil do
    begin
    sb.append(L);
    sb.append(#10);
    L:=Reader.readLine;
    end;
  Result:=sb.ToString;
end;
```

With the above 2 routines, we have all code needed to fetch data from an URL and convert it to a string. For a REST service, the string will typically contain JSON Data describing groups and pupils.

3 Threads or Asynchronous tasks

Depending on the speed of the internet connection and the amount of data, fetching and processing JSON can be a long operation. For this reason, the download is better done in a separate thread or an asynchronous task: the application will remain responsive during the execution of this task.

The Android API offers the `android.os.AsyncTask` class for this purpose. This is a generic class (in Java) and it is equivalent to the `TThread` class in Object pascal. The following is part of its (protected) interface:

```
function doInBackground(const para1: array of JLObject): JLObject;
procedure onPreExecute();
procedure onPostExecute(para1: JLObject);
procedure onProgressUpdate(const para1: array of JLObject);
```

All these methods are virtual, and the `doInBackground` is abstract, which means that a descendant class must be created which overrides at least this method .

Note: the released Free Pascal JVM compiler contains a bug that prevents the use of the above class as-is. The declaration of the class actually contains 2 overloaded versions of both `doInBackGround` and `onProgressUpdate`. To be able to use the class, the compiler bug must be worked around, this can be done by commenting out one of the `doInBackGround` methods (and likewise for `onProgressUpdate`).

Of the methods presented here, the `doInBackGround` method is the only one that is executed in a separate thread: it must be overridden (because it is an abstract method) and must contain the code that actually executes the background task.

The other methods are executed in the thread that created the asynchronous task. This is important: like most (if not all) GUI widgetsets, the Android GUI is not thread safe. That means that code in `doInBackground` cannot update the display. Updating the display can be done by overriding the `onPreExecute`, `onPostExecute` and `onProgressUpdate` methods: these methods are executed in the context of the main thread.

To use this class, we create the following descendent:

```
TOnSuccessHandler = Procedure(AResult : String) Of Object;

TDownloadHelperTask = Class(AOAsyncTask)
Protected
  function downloadUrl(AURL: String): String;
  function ReadAsString(ins: JIInputStream): String;
  Function doInBackground(const para1: array of JLObject): JLObject;
  Procedure onPostExecute(para1: JLObject); override;
Public
  Property OnSuccess : TOnSuccessHandler;
  Property OnError : TOnSuccessHandler;
end;
```

The properties `OnSuccess` and `OnError` are event handlers that will be called when the download was succesful or when an error happened - respectively. They will be executed in the `onPostExecute`, passing the JSON or an error message as the `AResult` parameter, depending on the result of the download.

The declaration contains the `downloadUrl` and `ReadAsString` methods discussed earlier. The `doInBackground` method must do the actual work, and is in fact very easy:

```
function TDownloadHelperTask.doInBackground(
         const para1: array of JLObject): JLObject;

Var
  S,Res : String;

begin
  Result:=Nil;
  FErrorMsg:='';
  Res:='';
  S:='';
  try
    S:=JLString(para1[0]);
    Res:=DownloadUrl(S);
    Result:=JLString.Create(Res);
  except
    On e : JLThrowable do
      FErrorMsg:='Unable to retrieve web page: URL may be invalid.';
  end;
end;
```

The Java declaration of the `doInBackground` method is in fact a Generic method. This is translated to Object Pascal using an `array of JLObject` parameter: the actual parameters to the `Execute` method of the `AsyncTask` are passed using this catch-all parameter: All types in Java are objects, hence all kinds of parameter can be passed.

In our class, only the URL must be passed on: it is then available in `para1[0]`. The return value of `Execute` and `doInBackGround` is an object. The result of `DownLoadURL` is converted to a string object and returned as the result.

The download routine is enclosed in a `try..except` block, so errors are caught: an error message variable is set. Failing to catch an error in a background task will kill a running application, so it is vital to catch errors.

The result of the `doInBackground` method is passed to the `onPostExecute` procedure, which will be executed in the main thread. The `onPostExecute` procedure simply calls the event handlers :

```
procedure TDownloadHelperTask.onPostExecute(para1: JLObject);

Var
  S : String;

begin
  S:=JLString(Para1);
  if (FErrorMsg<>'') then
    begin
    If Assigned(OnError) then
      OnError(FErrorMsg);
    end
  else If Assigned(OnSuccess) then
    OnSuccess(S);
end;
```

And with this, the asynchronous task class is finished.

4 Executing threads or Asynchronous tasks

Now that the class to execute code asynchronously is ready, we can use it in the GUI of our Android application. The main activity of the application is the `TGroupActivity`.

In a previous article this activity was configured with an options menu to show the preferences dialog. This menu can be expanded to show a 'Synchronization' menu item:

```
function TGroupActivity.onCreateOptionsMenu(AMenu: AVMenu): boolean;

begin
  inherited;
  AMenu.add(0,AVMenu.FIRST,0,R.strings.Preferences);
  AMenu.add(0,AVMenu.FIRST+1,0,R.strings.Synchronize);
  Result:=True;
end;
```

The Synchronize string must be added to the resoruces of the application, a process explained in the previous articles. When clicked, this menu item should start the synchronization process. It should do two things:

1. Import groups
2. Import pupils. This can only be done after the groups were imported.

These 2 tasks will be performed one after the other, but each is performed asyncronously. The menu is shown in figure 1 on page 8.

So; the `onOptionsItemSelected` onclick handler must be expanded to call the synchronization code, which is in a routine called `importgroups`. When the `ImportGroups` routine is finished, it must start the pupils import. During the import process, a progress dialog is shown.

The task to execute after the import of groups is passed on as an event handler (`DoNext`). It is saved for later use, and is executed when the groups import has finished.

For this simple case, the use of this variable is of course redundant, but it serves to illustrate a point: in more complex cases a series of procedures to be executed asynchronously can be constructed in such a manner.

The `ImportGroups` routine starts out with creating a dialog, which is shown during the download and the processing of the response. The progress dialog is a standard dialog of Android: `android.app.ProgressDialog`. It has several interesting methods:

setIndeterminate The dialog will show a wait animation if called with True as an argument.

setMessage Sets the message that is displayed in the dialogue.

hide hides the dialog, but keeps it in memory.

dismiss Dismisses the dialog (hides it and removes it from memory).

setCancelable The `setCancelable` method can be used to determine whether the user can cancel the dialog. This would require canceling the thread and import process: we will not implement this in this example.

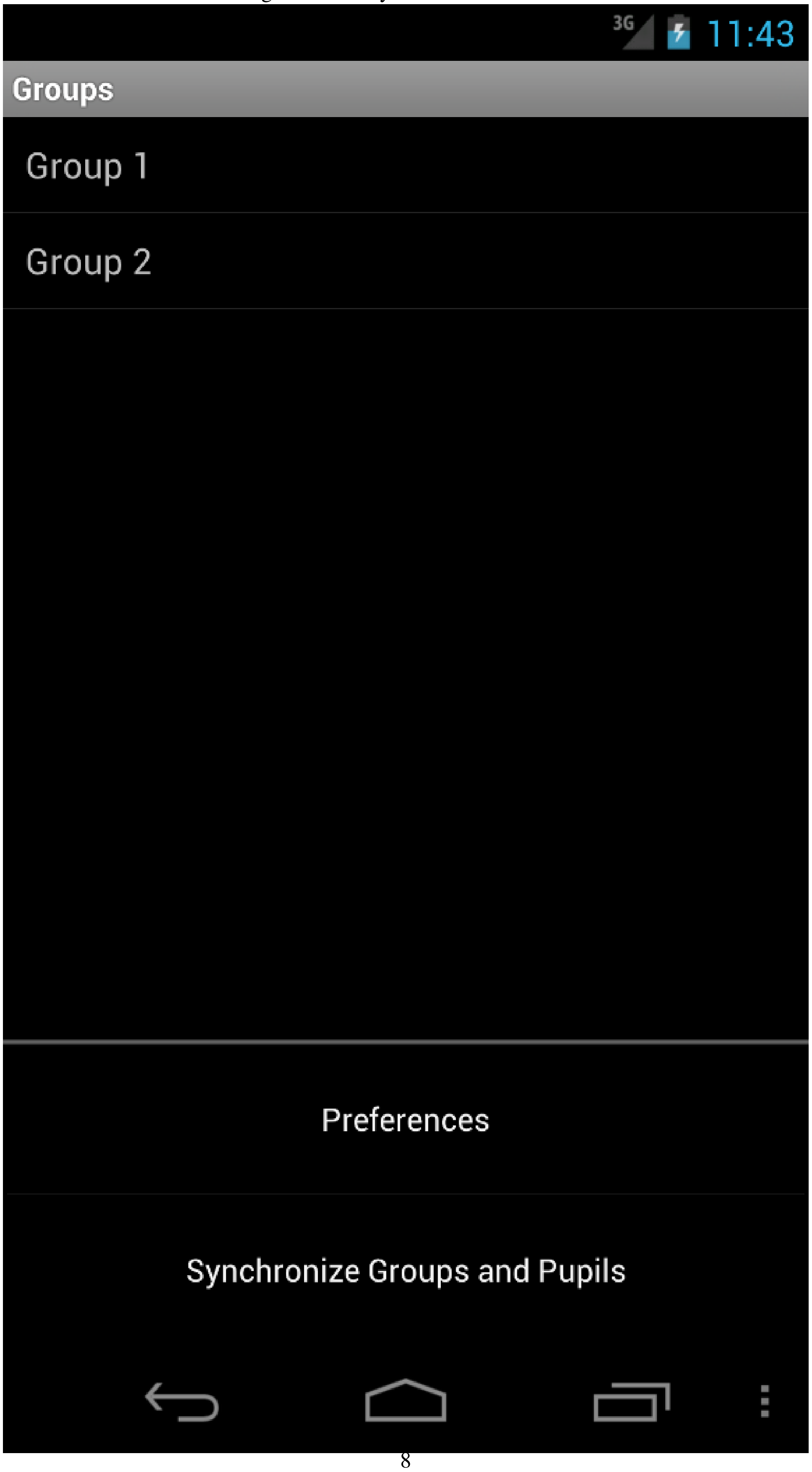
using these methods, a dialog (`Fimporting`) is shown, and the asynchronous task is launched:

```
Procedure TGroupActivity.ImportGroups(DoNext : TNextProcedure);

Const
  MyURL : 'http://192.168.0.98:60080/~michael/groups.json';
Var
  URL : JLString;

begin
  FDoNext:=DoNext;
  FImporting:=AAProgressDialog.create(Self);
  FImporting.setMessage(JLString.Create('Fetching groups'));
  FImporting.setIndeterminate(True);
  FImporting.setCancelable(False);
  FImporting.show();
  FImportTask:=TDownloadHelperTask.Create();
```

Figure 1: The synchronization menu



```
  FImportTask.OnError:=@CancelImport;
  FImportTask.OnSuccess:=@SaveNewGroups;
  URL:=JLString.Create(MyURL);
  FImportTask.execute([URL]);
end;
```

An instance of the task (`FImportTask`) is created and configured by setting the event handlers for success (to save the new groups) and failure (cancel the import). After which its `Execute` method is called: the `Execute` method is passed the URL of the JSON data to download.

Additional parameters could for instance include an instance of the progress dialog, so it can be updated in the `onProgressupdate` method. In the example, the URL is hard-coded but in a real-world application, this would probably be configurable or set to an actual web service endpoint.

The `CancelImport` method is called when an error happens during download. There is little special about this function, other than that it demonstrates another way of creating a dialog: It clears the import task reference (the java runtime will garbage collect the object) and then dismisses the progress dialog (`FImporting`) and constructs a new dialog `Android.app.AlertDialog`. The dialog is constructed using a `builder` class:

```
Procedure TGroupActivity.CancelImport(S : String);

Var
  Bld : AAAlertDialog.InnerBuilder;
  Dlg : AAAlertDialog;
  JS : JLString;

begin
  FImportTask:=Nil;
  If FImporting<>Nil then
    FImporting.dismiss();
  Bld:=AAAlertDialog.InnerBuilder.Create(self);
  JS:=JLString.Create('Synchronization failed: '+S);
  Bld.setMessage(JS);
  Bld.setCancelable(true);
  Dlg:=Bld.create_();
  Dlg.show();
end;
```

The builder class has a multitude of methods which add various elements to a dialog or set particular properties of the new dialog. Only 3 methods are used here:

setMessage this method sets the message that is displayed in the dialog.

setCancelable Passing `True` to this method allows the user to cancel (and close) the dialog.

create_ This function is not exactly a constructor, but it does create a dialog instance with all the properties set according to how the various methods were called, and then returns the dialog instance.

The separate instance of `JLString` (`JS`) is needed, because the `setMessage` method expects a `JSCharsequence` instance (of which JLString is a descendant). In this case the automatic correspondence of Object Pascal strings and Java strings is not automatically available, and a string must be explicitly constructed.

The SaveNewGroups method is called when the download of JSON data has succeeded:

```
Procedure TGroupActivity.SaveNewGroups(S : String);

begin
  FImportTask:=Nil;
  if (S='') or (Pos('{',S)<>1) then
    CancelImport('No valid groups JSON from server')
  else
    begin
    FImporting.SetMessage(JLString.Create('Saving groups'));
    try
      // update database
      fDataHelper.ImportGroups(S);
      // update display
      FillItems(False);
    except
      CancelImport('An error occurred while saving the groups');
      exit;
    end;
    if Assigned(FDoNext) then
      FDoNext;
    end;
end;
```

In case a wrong document arrived from the server, or there was an error during the saving of the groups, the CancelImport is called, which will dismiss the progress dialog and shows an error message instead.

If the import went well, the FDoNext procedure is called. This is normally set to the ImportPupils method, which does the same as import groups, but fetches a different URL, and calls fDataHelper.ImportPupils.

5 Parsing JSON using external Java classes

The fpJSON unit distributed with Free Pascal is not yet available for the Android target. Therefor, parsing JSON must be done using some other means. Luckily a lot of JSON classes exist in Java.

In this article, we'll take a very simple implementation by Douglas Crockford, available on Github:

```
https://github.com/douglascrockford/JSON-java
```

Downloading these Java classes provides us with several Java files. Not all files are needed. Only the JSONString, JSONObject, JSONException JSONTokener and JSONArray java files are needed. They must be compiled using the java compiler, and then object pascal import classes can be generated from the compiled files.

The JSON java files can be compiled with the 1.7 Java compiler. The following assumes that the Java SDK is installed, and that the Java compiler javac and jar commands are present on your computer, and that they are available through the path.

The above files must be compiled by the Java compiler. This can be done on the command-line using the following command:

```
javac -d . JSONException.java JSONString.java \
  JSONTokener.java JSONObject.java JSONArray.java
```

This will create a folder org/json, containing the compiled java classes. The org folder must be copied to the folder where the absentee application is stored (more on this later).

The compiled classes can be put in a .jar file with the following command:

```
jar cf json.jar org
```

The contents of the jar file can be examined easily: simply change the extension to .zip, and examine the contents in the windows explorer...

Now that we have a jar file, we can create Object Pascal classes from it using the JavaPP tool that is distributed with Free Pascal. This tool is a Java tool, so you need the Java runtime to work with it. The following command will convert the .jar to a pascal file (notice the dot at the end):

```
javapp -o jsonobject -classpath json.jar org/json.
```

This will create a unit jsonobject.pas and jsonobject.inc. This unit can be added to the abseenteeapp project. The following Object Pascal classes now become available:

OJJSONObject A class representing a JSON object, corresponds roughly to `TJSONObject` in fpJSON.

OJJSONArray A class representing a JSON array, corresponds roughly to `TJSONArray` in fpJSON.

OJJSONString A class representing a JSON string, corresponds roughly to `TJSONString` in fpJSON.

These classes can now be used to implement the `ImportPupils` and `ImportGroups` routines that were called after the JSON was downloaded. The `ImportGroups` routine is coded to consume JSON data of the form:

```
{ "data" : [
    { "ID" : 1, "Name": "Group 1" },
    { "ID" : 2, "Name": "Group 2" }
  ]
}
```

This is easily implemented as follows:

```
procedure TAbsenteeDBHelper.ImportGroups(AJSON: String);

Var
  j,p : OJJSONObject;
  a : OJJSONArray;
  I : Integer;
  DB : ADSSqliteDatabase;

begin
  J:=OJJSONObject.Create(AJSON);
  a:=J.getJSONArray('data');
  if A.Length>0 then
```

```
    begin
    DB:=GetWritableDatabase;
    for i:=0 to a.length-1 do
      begin
      P:=a.getJSONObject(i);
      ImportGroup(DB,P.getInt('ID'),P.GetString('Name'));
      end;
    end;
end;
```

This routine does not look very different from how it would be coded using fpJson objects: getting the data array, and then a simple loop over the array.

The `ImportGroup` routine performs the task of inserting the record in the database. It does not insert a group that already exists: checking for the existence of the group is done by searching for the remote ID of the group in the local database (using a function `GetLocalGroupID`, not presented here).

```
procedure TAbsenteeDBHelper.ImportGroup(DB : ADSSqliteDatabase;
                                        AID : Integer;
                                        AName : string);

Var
  Q : String;
  gid : Integer;
  SID : String;

begin
  gid:=GetLocalGroupID(DB,AID);
  if Gid=-1 then
    begin
    SID:=JLLong.Create(AID).toString;
    Q:='insert into groups (gr_remote_id, gr_name)';
    Q:=Q+' values ('+SID+','''+AName+''')';
    DB.ExecSQL(Q);
    end;
end;
```

The remote ID is stored in the database, so that when the user synchronizes the groups and pupils again, the routine detects which groups have already been imported. Deleting groups that no longer exist on the server can also be done, but this is left as an exercise to the reader.

Similar routines can be created for the import of pupils, they consume JSON of the form:

```
{ "data" : [
  {"ID": 123, "FirstName": "Michael",
   "LastName": "Van Canneyt", "GroupID": 1},
  { "ID" : 345, "FirstName": "Jonas",
    "LastName": "Maebe", "GroupID"  : 1}
] }
```

Similarly to the import of pupils, the remote ID is stored locally and used to check existence of the pupil in the local database. Each pupil can be a member of only one group with this JSON structure. The code to import this will not be presented here, as it is very similar to the code for groups.

6 Packaging the application

To create an installable package (apk), we must ensure that the JSON classes are also packaged. Fortunately, there is in fact nothing to be done extra for this, except copy the output files from the Java compiler (used to compile the json classes) to the bin/classes folder below the absentee app. The dex compiler (used to transcode ordinary Java bytecode to bytecode usable in the Dalvik VM) will then pick up the JSON classes and add them to the classes.dex file it generates for the application.

In order for the application to be able to access the network, the following `uses-permission` tag must also be added to the AndroidManifest.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
   xmlns:android="http://schemas.android.com/apk/res/android"
   package="eu.blaisepascal.absenteeapp"
   android:versionCode="1"
   android:versionName="1.0">
   <uses-permission
       android:name="android.permission.INTERNET"/>
```

If this is omitted, a “permission denied” error will occur when the application attempts to download files from internet. If the application must be able to start WIFI (using the `TAbsenteeDBHelper.EnableWIFI` method presented above), an additional `uses-permission` tag must be added to the manifest file:

```
<uses-permission
    android:name="android.permission.CHANGE_WIFI_STATE" />
```

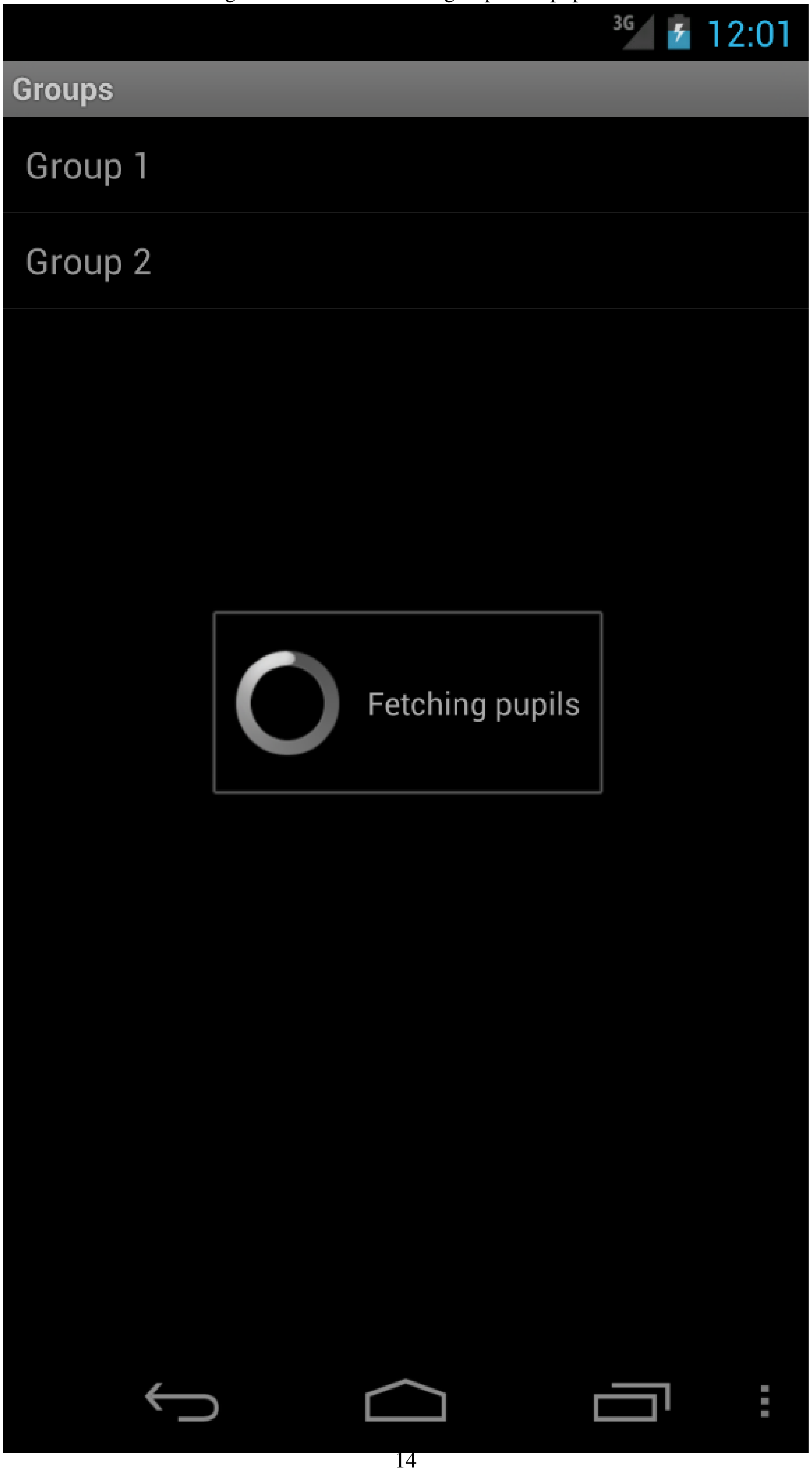
When all this is done and the application is started, pressing the ’Synchronize groups and Pupils’ option menu will result in an import of groups and pupils, during which an image as in figure 2 on page 14 will be displayed.

7 Conclusion

In this article, we’ve shown how to execute HTTP requests in the background using an AsyncTask, and how to process the JSON content of these requests using external Java classes. The import of JSON classes was more an exercise in handling of external classes than a necessity: the Android API already contains the same set of JSON classes, and they are accessible through the standard Androidr14 unit.

In the next article, this newly gained knowledge will be combined with some new techniques (file and directory access) to download pictures from a website and show each pupil’s avatar - when available - next to his name in the listview: After all, that is a standard practice when programming mobile apps, and there is no reason why it should not be possible in Object Pascal.

Figure 2: The screen during import of pupils



3G
12:01
Groups
Group 1
Group 2
Fetching pupils